**1️ What is Inner and Outer Sequence in LINQ?**

* **Outer Sequence**: This is the first collection in GroupJoin. It provides the main dataset.
* **Inner Sequence**: This is the second collection in GroupJoin. It contains data related to the outer sequence.

**🔹 What is an Outer Join?**

An **Outer Join** is a type of join where all records from one table (outer table) are returned, even if there is no matching record in the other table (inner table). If there's no match, NULL values are returned for columns of the inner table.

There are two types of Outer Joins:

1. **Left Outer Join**: All records from the left table (outer sequence) are included, even if there's no match in the right table (inner sequence).
2. **Right Outer Join**: All records from the right table (inner sequence) are included, even if there's no match in the left table (outer sequence).

**🔹 Implementing Left Outer Join in LINQ**

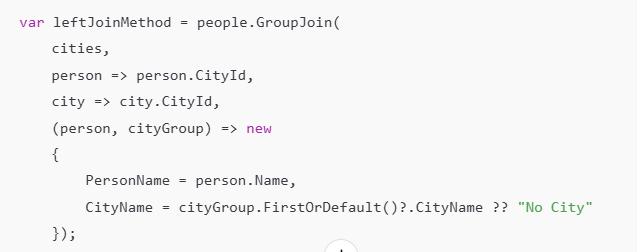
In LINQ, **Left Outer Join** is the most common use of Outer Join. It ensures that if there's no match from the inner sequence, it will still include the outer sequence and return null for the missing values.

A **Right Outer Join** ensures that all records from the right table (the inner sequence) are included in the result, even if there is no matching record in the left table (the outer sequence). If there's no match, the fields from the left table (outer sequence) will be null.

In LINQ, implementing a **Right Outer Join** is a little trickier than the **Left Outer Join** because we typically group by the outer sequence, but in a right join, we need to consider the inner sequence (the right sequence) as the starting point and ensure it’s included.

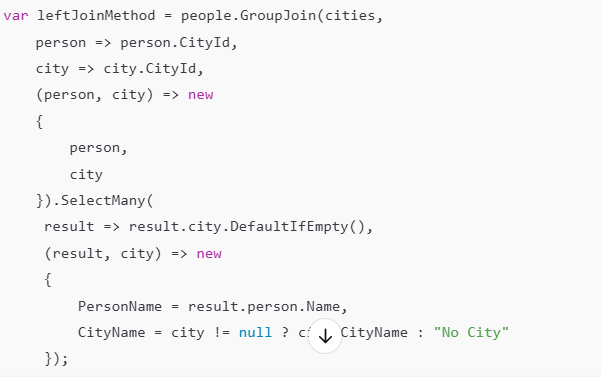
The two leftJoinMethod code snippets you've provided are different ways to perform a left join using LINQ, but they are structured differently in terms of how they handle the joining and how they manage the cities with no corresponding people (or vice versa). Let's go through each one to highlight the differences.

**1. First leftJoinMethod (Using GroupJoin and FirstOrDefault):**



* **GroupJoin**: Here, the GroupJoin is performed where you are grouping people based on their CityId and matching them with CityId from the cities list.
* **Handling Missing Cities**:
  + FirstOrDefault() is used to pick the first city in each group (for a person that has a city). If there is no match (i.e., a person has no city), FirstOrDefault() returns null.
  + The ?. (null conditional operator) ensures that if FirstOrDefault() returns null, the code will safely return "No City" as the fallback value.
* **Outcome**: This approach will result in one entry per person. For people with no city, it will display "No City". The result is a collection where each person is paired with a city or "No City" if there is no city.

**2. Second leftJoinMethod (Using GroupJoin, SelectMany, and DefaultIfEmpty):**



* **GroupJoin**: This part is similar to the first method. It groups people by CityId and matches them with cities by CityId.
* **Handling Missing Cities**:
  + SelectMany flattens the GroupJoin result. Normally, a GroupJoin produces a collection of cities for each person. To handle cases where there is no matching city (for a person), DefaultIfEmpty() is called on result.city.
  + DefaultIfEmpty() ensures that for people with no matching cities, the collection will still contain a null value (indicating no city), instead of being empty.
  + In the final projection (result, city) => new { ... }, the city value will either contain the matched city or be null. If city is null, "No City" is returned.
* **Outcome**: This approach also results in one entry per person, but it works by flattening the collection using SelectMany, which gives you the same result but with a slightly different flow.

**Key Differences:**

1. **Approach to Handling No Matches**:
   * **First Method**: Uses FirstOrDefault() to grab the first city (or null if no match) and then handles null by using the null-conditional operator (?.), followed by a fallback string "No City".
   * **Second Method**: Uses SelectMany to flatten the result of GroupJoin and DefaultIfEmpty() to ensure that people without a corresponding city still get included. The DefaultIfEmpty() ensures there is a null city when no matches are found, and the final projection handles the null by returning "No City".
2. **Performance**:
   * **First Method**: This method performs a single-level join with GroupJoin and selects the first city for each person. If a person has no city, it assigns "No City". This might be slightly more efficient because it doesn't flatten the result like SelectMany does.
   * **Second Method**: This method uses SelectMany to flatten the results, which means it creates a "flat" list where each person is paired with a city (or null if no city exists), then the city is checked for null to assign "No City". This is a bit more complex but can be useful when you need to work with the results as a flattened sequence.
3. **Flattening**:
   * **First Method**: The result is a sequence of anonymous objects, where each person is paired with a single city name or "No City".
   * **Second Method**: The result is a sequence of anonymous objects as well, but with more flexibility in terms of handling the join and flattening the result.
4. **Output Structure**:
   * Both approaches will ultimately produce the same output for your data. The output will include each person's name and the name of their city (or "No City" if there is no city). However, the intermediate steps (and the LINQ methods used) differ in how they produce that output.

**Conclusion:**

* **Use the first method** (GroupJoin with FirstOrDefault() and null-conditional operator) when you simply need to match each element of the outer sequence (people) with the first matching element from the inner sequence (cities), and want to provide a fallback value when no match is found.
* **Use the second method** (GroupJoin with SelectMany and DefaultIfEmpty()) when you want to flatten the grouped results, ensuring that you get a sequence where every person is matched with a city (or null if there is no city), and then need to handle the null values.

Both methods are valid for performing a left join, but the second method is more flexible and suitable for more complex scenarios involving flattened results.

# Why Union needs to use FirstOrDefault?

The issue with using Union in your second approach stems from how LINQ handles anonymous types and the Distinct method (which Union uses). Let's break down why Union doesn't work directly with your second approach.

**Understanding the Problem**

When you use GroupJoin with SelectMany and DefaultIfEmpty(), you are creating a **flattened sequence of anonymous objects**. This anonymous object contains two properties: PersonName and CityName. Here's what happens when you try to use Union in this scenario:

* Union requires that the two sequences you're combining have the **same type** (specifically, the anonymous type in this case). However, when you're performing a GroupJoin and using SelectMany, you're dealing with a sequence of anonymous types that may differ slightly, especially in terms of their order or structure, even though they are functionally similar.
* The anonymous types you create in each of the join methods (left join, right join, etc.) are distinct, and even though they have the same fields (PersonName and CityName), they are **not exactly the same** in terms of how LINQ compares them internally.

**Explanation of Why Union Fails:**

1. **Distinctness of Anonymous Types**: When you perform a Union, it compares the items in the two sequences based on their structure. Since the structure of the two anonymous types generated in the left join and right join operations is very slightly different, the Union method sees them as **different types**, even though they contain the same property names.
2. **Anonymous Types Are Not Easily Reused**: Even if the properties have the same names, the compiler treats each anonymous type as a **distinct type**. So, if you attempt to combine the results of two operations that yield anonymous types, LINQ can't always determine that they should be treated as the same type, leading to issues when you try to use Union.

**Solution**

To fix this issue, you can **cast** the anonymous types into a defined object or structure (a named class or struct). This way, LINQ can compare the types properly, and Union will work because both sequences are now of the same type.

**How to Fix:**

1. **Define a Custom Class** for the result:

Instead of using an anonymous type in your join queries, define a class to represent the result. For example:

public class PersonCity

{

public string PersonName { get; set; }

public string CityName { get; set; }

}

1. **Modify your queries** to return instances of PersonCity:
2. var leftJoinMethod = people.GroupJoin(
3. cities,
4. person => person.CityId,
5. city => city.CityId,
6. (person, cityGroup) => new PersonCity
7. {
8. PersonName = person.Name,
9. CityName = cityGroup.FirstOrDefault()?.CityName ?? "No City"
10. });
11. var rightJoinMethod = cities.GroupJoin(
12. people,
13. city => city.CityId,
14. person => person.CityId,
15. (city, peopleGroup) => new PersonCity
16. {
17. PersonName = peopleGroup.FirstOrDefault()?.Name ?? "No Person",
18. CityName = city.CityName
19. });
20. // Now Union will work because both queries return the same type
21. var fullOuterJoin = leftJoinMethod
22. .Union(rightJoinMethod)
23. .Distinct();
24. **Use Union to combine the sequences**:

Now that both the leftJoinMethod and rightJoinMethod return PersonCity instances, Union will work correctly, as both sequences are of the same type (PersonCity).

**Why This Works:**

By creating a named type (PersonCity), both the left join and right join methods return sequences of the same type, and Union can now correctly combine the sequences, removing duplicates based on the properties of the PersonCity class. Additionally, the compiler no longer treats the results as distinct anonymous types, which solves the problem you were facing.

**Summary of Changes:**

* **Anonymous Types**: LINQ treats anonymous types as distinct, which causes problems when you try to use Union or Distinct.
* **Solution**: Define a class (like PersonCity) and make sure both join queries return that same type.
* **Outcome**: After defining a custom class, Union will work as expected because both sequences are now of the same type.